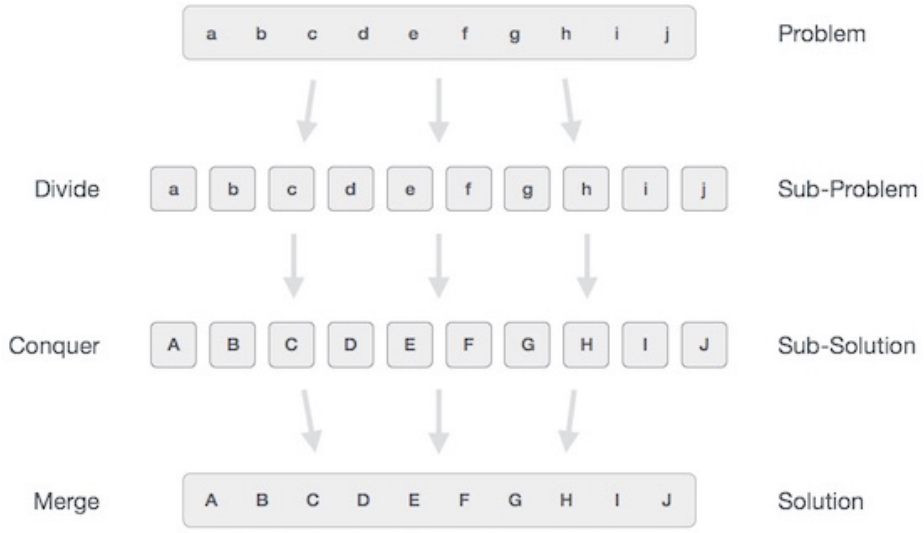
UNIT II

DIVIDE AND CONQUER

Introduction, Binary Search - Merge sort and its algorithm analysis - Quick sort and its algorithm analysis - Strassen's Matrix multiplication - Finding Maximum and minimum - Algorithm for finding closest pair - Convex Hull Problem

INTRODUCTION

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of original problem.



Broadly, we can understand **divide-and-conquer** approach as three step process.

Divide/Break

- This step involves breaking the problem into smaller sub-problems. Sub-problems should represent as a part of original problem. This step generally takes recursive approach to divide the problem until no sub-problem is further dividable. At this stage, sub-problems become atomic in nature but still represents some part of actual problem.

Conquer/Solve

- This step receives lot of smaller sub-problem to be solved. Generally at this level, problems are considered 'solved' on their own.

Merge/Combine

- When the smaller sub-problems are solved, this stage recursively combines them until they formulate solution of the original problem.

Examples

There are various ways available to solve any computer problem but the following computer algorithms are based on **divide-and-conquer** programming approach –

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest pair (points)

BINARY SEARCH ALGORITHM

Binary search algorithm finds given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in an order. The binary search can not be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list". Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matching, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

- **Step 8:** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Pseudo code

```
binarysearch(a[n], key, low, high)
while(low<high)
{
mid = (low+high)/2;
if(a[mid]=key)
     return mid;
else if (a[mid] > key)
     high=mid-1;
  else
     low=mid+1;
  }
return -1;
```

Program

```
#include <stdio.h>
int main()
{
int first, last, middle, size, i, sElement, list[100];
   printf("Enter the size of the list: ");
   scanf("%d",&size);
   printf("Enter %d integer values in ascending order\n", size);
   for (i = 0; i < size; i++)
      scanf("%d",&list[i]);
   printf("Enter value to be search: ");
   scanf("%d", &sElement);
   first = 0;
   last = size - 1;
   middle = (first+last)/2;

   while (first <= last) {
      if (list[middle] < sElement)
         first = middle + 1;
      else if (list[middle] == sElement) {
         printf("Element found at index %d.\n",middle);
         break;
      }
      else
```

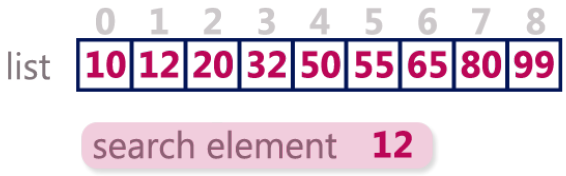
```
         last = middle - 1;

      middle = (first + last)/2;
   }
   if (first > last)
      printf("Element Not found in the list.");
  return 0;
}
```

Sample input and output:

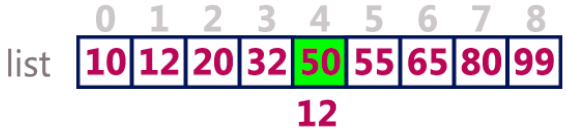
```
Enter the size of the list:4
Enter 4 integer values in ascending order 4 7 9 11
Enter value to be search: 9
Element found at index 2
```

Example: Consider the following list of elements and search element 12



Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

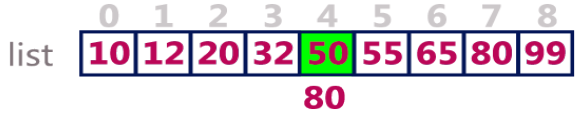
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

search element 80

Step 1:

search element (80) is compared with middle element (50)

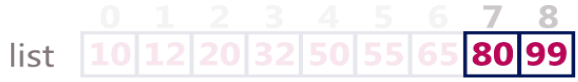


Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).



Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Complexity Analysis

Worst case analysis: The key is not in the array

Let T(n) be the number of comparisons done in the worst case for an array of size n. For the purposes of analysis, assume n is a power of 2, i.e. n = 2^k.

Then $T(n) = 2 + T(n/2)$

$= 2 + 2 + T(\frac{n}{2^2})$ // 2nd iteration

$= 2 + 2 + 2 + T(n/2^3)$ // 3rd iteration

...

$= i * 2 + T(n/2^i)$ // ith iteration

... $= k * 2 + T(1)$

Note that k = logn, and that T(1) = 2.

So T(n) = 2logn + 2 = O(logn)

So we expect binary search to be significantly more efficient than linear search for large values of n.

MERGE SORT

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

Steps

1. Input the total number of elements that are there in an array (number_of_elements).
2. Input the array (array[number_of_elements]).
3. Then call the function MergeSort() to sort the input array. MergeSort() function sorts the array in the range [left,right] i.e. from index left to index right inclusive.
4. Merge() function merges the two sorted parts. Sorted parts will be from [left, mid] and [mid+1, right]. After merging output the sorted array.

Algorithm

Sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (*A*, 1, *n*).

```
MERGE-SORT (A, p, r)
1.   WHILE p < r                                  // Check for base case
2.         q = FLOOR[(p + r)/2]          // Divide step
3.          MERGE-SORT (A, p, q)
4.                MERGE-SORT (A, q + 1, r)
5.                 MERGE (A, p, q+1, r)
```

MergeSort() function / Split step

For *mergesort* function we get three parameters, the input array *a[], the start index and the end* index of array. This start and end change in every recursive invocation of mergesort function. We find the middle index using start and end index of the input array and again invoke the same function with two parts one starting from start to mid and other being from mid+1 to end. Once base condition is hit, we start winding up and call merge function. *Merge* function takes four parameters, *input array, start, end and middle index* based on start and end.

Merge() function / Merge step

Sort the smallest parts and combine them together with merge operation. In merge operation, scan both sorted arrays and based on the comparison, put one of the two items into output array, till both arrays are scanned. If one array is finished before other, just copy all elements from the other array to output array. Copy this output array back to original input array so that it can be combined with bigger sub problems till solution to original problem is derived.

Merge function merges two sub arrays (one from start to mid and other from mid+1 to end) into a single array from start to end. This array is then returned to upper calling function which then again sort two parts of array.

The **pseudocode** of the MERGE procedure is as follow:

MERGE (A, p, q, r)

```
1.  n1 ← q − p + 1
2.  n2 ← r − q
3.  Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
4.  FOR i ← 1 TO n1
5.      DO L[i] ← A[p + i − 1]
6.  FOR j ← 1 TO n2
7.      DO R[j] ← A[q + j ]
8.  L[n1 + 1] ← ∞
9.  R[n2 + 1] ← ∞
10. i ← 1
11. j ← 1
12. FOR k ← p TO r
13.     DO IF L[i ] ≤ R[ j]
14.         THEN A[k] ← L[i]
15.              i ← i + 1
16.         ELSE A[k] ← R[j]
17.              j ← j + 1
```

Properties:

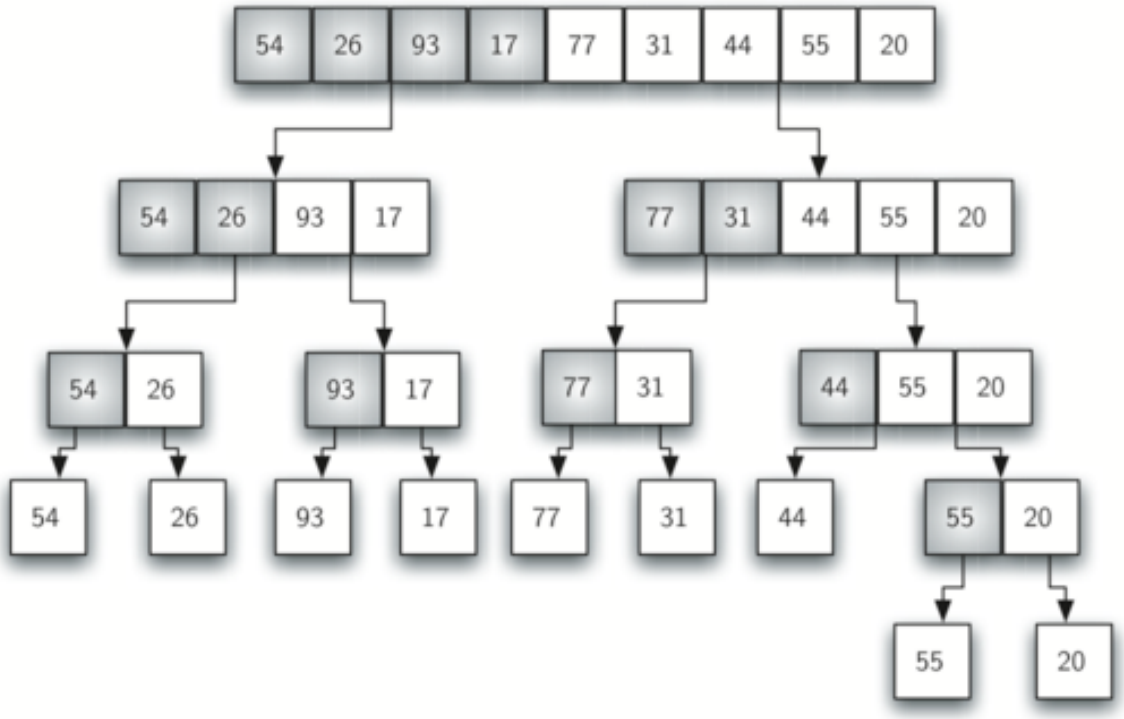
Best case – When the array is already sorted O(nlogn).

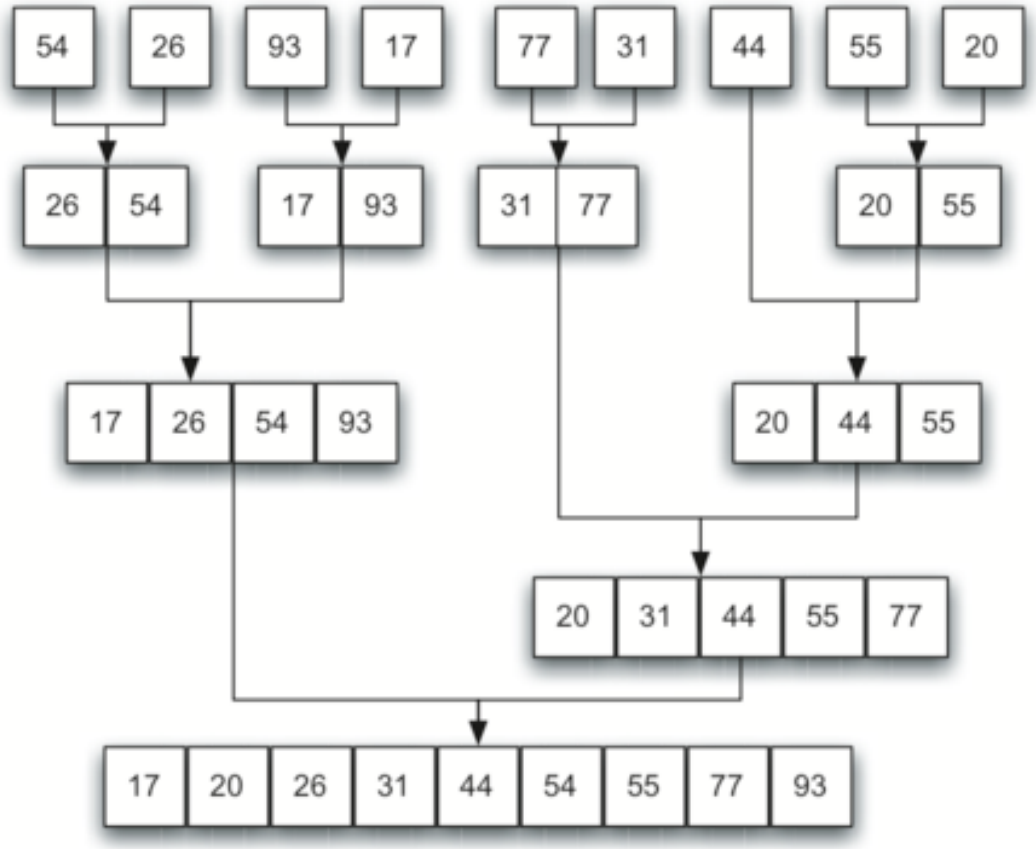
Worst case – When the array is sorted in reverse order O(nlogn).

Average case – O(nlogn).

Extra space is required, so space complexity is O(n) for arrays and O(logn) for linked lists.

Example


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20


54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

Example for merge

Complexity Analysis

Every time input space is divided into two, this division will have complexity of O(log n) where n is input size. First part or split part of implementation of merge sort has complexity of O(logn). Now the second part of implements merge step which place every element in its proper place in array, hence it linear time O(n). Since above step of dividing has to be done for n elements, hence total complexity of merge sort will be O(nlogn).

Analyzing Merge Sort

For simplicity, assume that n is a power of 2 so that each divide step yields two subproblems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

- **Divide**: Just compute q as the average of p and r, which takes constant time i.e. $\Theta(1)$.
- **Conquer**: Recursively solve 2 subproblems, each of size $n/2$, which is $2T(n/2)$.
- **Combine**: MERGE on an n-element subarray takes $\Theta(n)$ time.

Summed together they give a function that is linear in n, which is $\Theta(n)$. Therefore, the recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solving the Merge Sort Recurrence

By the master theorem in CLRS-Chapter 4 (page 73), we can show that this recurrence has the solution

$T(n) = \Theta(n \lg n)$.

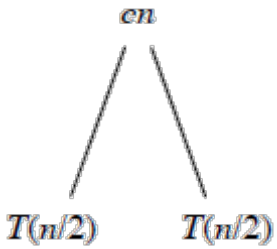
Reminder: $\lg n$ stands for $\log_2 n$.

Compared to insertion sort [$\Theta(n^2)$ worst-case time], merge sort is faster. Trading a factor of n for a factor of $\lg n$ is a good deal. On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sorts.

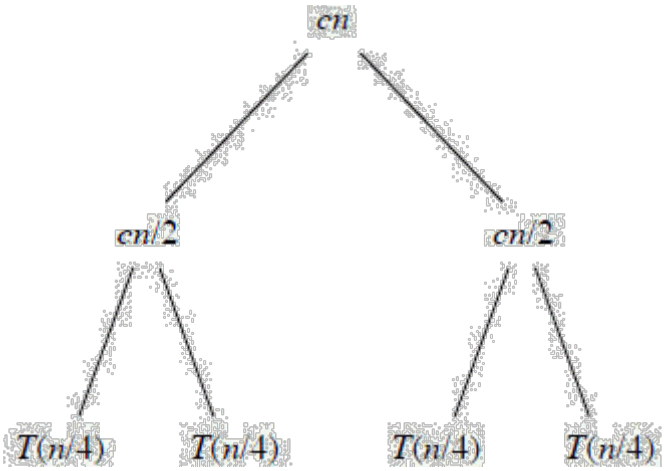
Recursion Tree

We can understand how to solve the merge-sort recurrence without the master theorem. There is a drawing of recursion tree on page 35 in CLRS, which shows successive expansions of the recurrence.

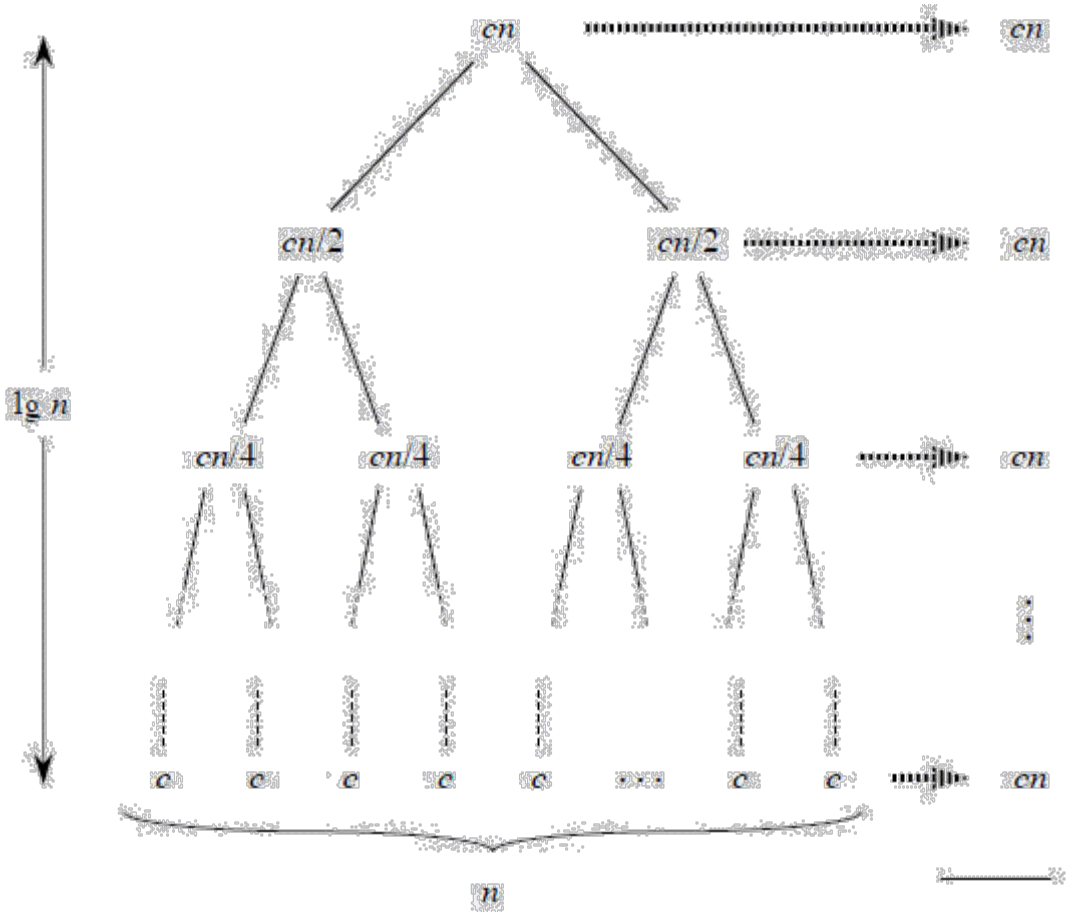
The following figure (Figure 2.5b in CLRS) shows that for the original problem, we have a cost of cn, plus the two subproblems, each costing $T(n/2)$.



The following figure (Figure 2.5c in CLRS) shows that for each of the size-$n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing T ($n/4$).



The following figure (Figure: 2.5d in CLRS) tells to continue expanding until the problem sizes get down to 1.



In the above recursion tree, each level has cost cn.

- The top level has cost cn.

- The next level down has 2 subproblems, each contributing cost $cn/2$.
- The next level has 4 subproblems, each contributing cost $cn/4$.
- Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves. Therefore, cost per level stays the same.

The height of this recursion tree is lg n and there are lg n + 1 levels.

Mathematical Induction

We use induction on the size of a given subproblem n.

Base case: $n = 1$

Implies that there is 1 level, and lg 1 + 1 = 0 + 1 = 1.

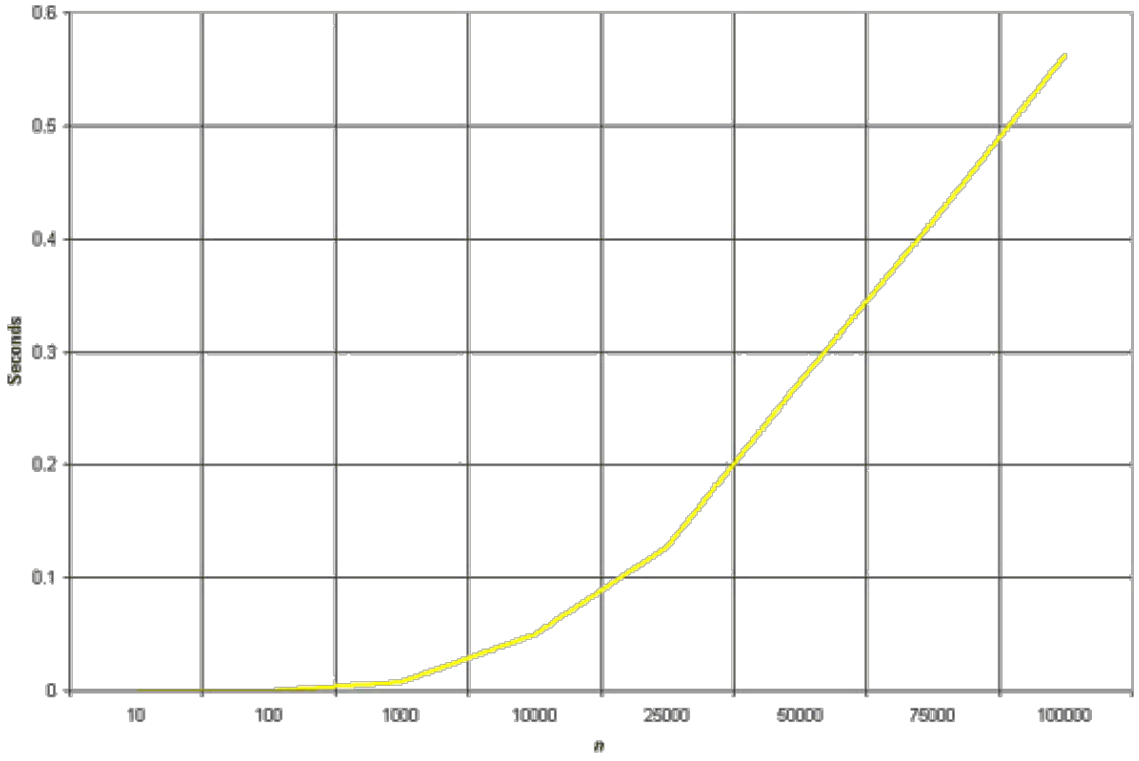
Inductive Step

Our inductive hypothesis is that a tree for a problem size of 2^i has lg 2^i + 1 = i +1 levels. Because we assume that the problem size is a power of 2, the next problem size up after 2^i is 2^i + 1. A tree for a problem size of 2^i + 1 has one more level than the size-2^i tree implying i + 2 levels.

Since lg 2^i + 1 = i + 2, we are done with the inductive argument.

Total cost is sum of costs at each level of the tree. Since we have lg n +1 levels, each costing cn, the total cost is cn lg n + cn.

Ignore low-order term of cn and constant coefficient c, and we have, $\Theta(n \lg n)$ which is the desired result.



QUICK SORT AND ITS ALGORITHM ANALYSIS

Quick sort works by partitioning a given array $A[p \,..\, r]$ into two non-empty sub array $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ such that every key in $A[p \,..\, q]$ is less than or equal to every key in $A[q+1 \,..\, r]$. Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

QuickSort ()

```
1.   If p < r then
2.            Partition (A, p, r)
3.   Recursive call to Quick Sort (A, p, q)
4.   Recursive call to Quick Sort (A, q + r, r)
```

Note that to sort entire array, the initial call Quick Sort (A, 1, length[A])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partitioning the Array

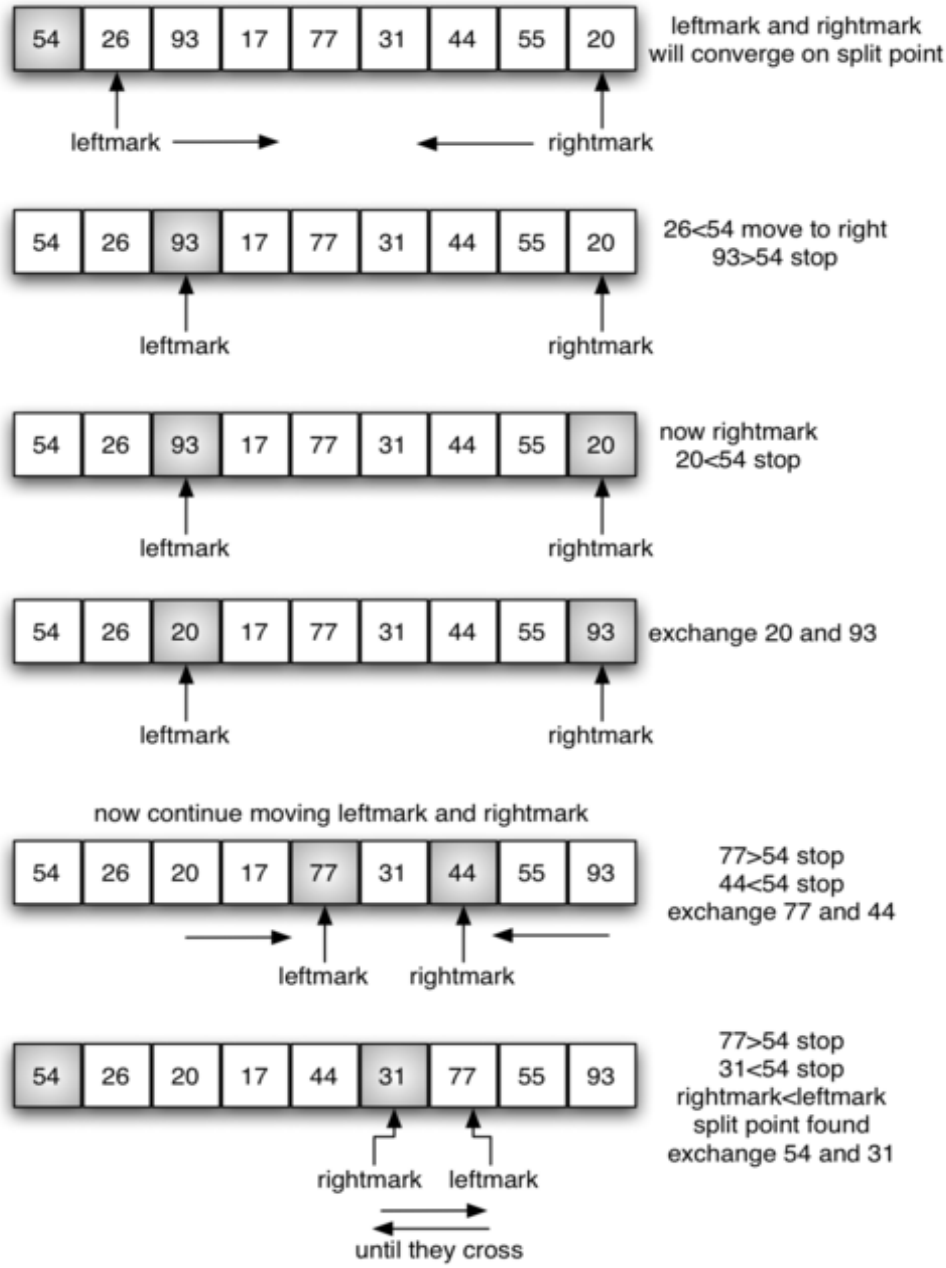
Partitioning procedure rearranges the sub arrays in-place.

PARTITION (A, p, r)

```
1.   x ← A[p]
2.   i ← p-1
3.   j ← r+1
4.   while TRUE do
5.       Repeat j ← j-1
6.       until A[j] ≤ x
7.       Repeat i ← i+1
8.       until A[i] ≥ x
9.       if i < j
10.          then exchange A[i] ↔ A[j]
11.          else return j
```

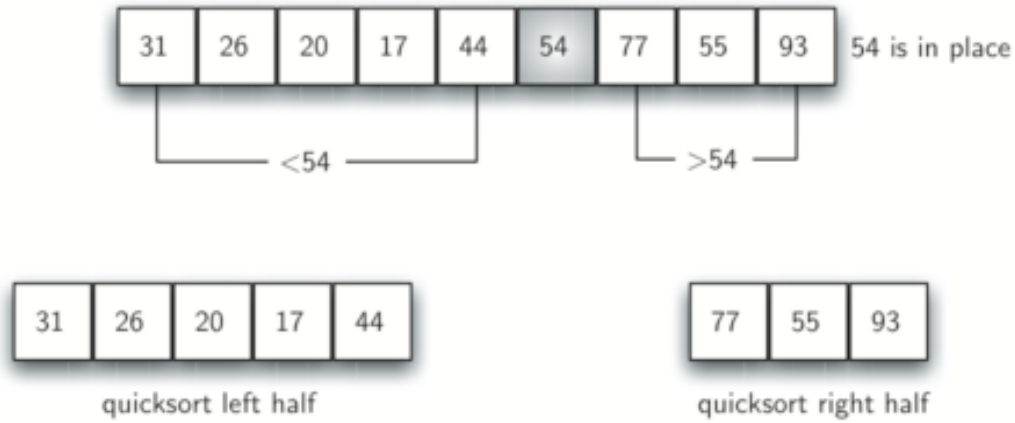
Example :

Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. This process as we locate the position of 54.



We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place. In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



Implementation in C

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than specified value say pivot based on which the partition is made and another array holds values greater than pivot value.

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count){
   int i;

   for(i = 0;i <count-1;i++){
```

```
      printf("=");
   }

   printf("=\n");
}

void display(){
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i<MAX;i++){
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void swap(int num1, int num2){
   int temp = intArray[num1];
   intArray[num1] = intArray[num2];
   intArray[num2] = temp;
}

int partition(int left, int right, int pivot){
   int leftPointer = left -1;
   int rightPointer = right;

   while(true){

      while(intArray[++leftPointer] < pivot){
         //do nothing
      }

      while(rightPointer > 0 && intArray[--rightPointer] > pivot){
         //do nothing
      }

      if(leftPointer >= rightPointer){
         break;
      }else{
         printf(" item swapped :%d,%d\n",
         intArray[leftPointer],intArray[rightPointer]);
         swap(leftPointer,rightPointer);
      }

   }

   printf(" pivot swapped :%d,%d\n", intArray[leftPointer],intArray[right]);
   swap(leftPointer,right);
   printf("Updated Array: ");
   display();
   return leftPointer;
}

void quickSort(int left, int right){
   if(right-left <= 0){
      return;
   }else {
      int pivot = intArray[right];
      int partitionPoint = partition(left, right, pivot);
      quickSort(left,partitionPoint-1);
      quickSort(partitionPoint+1,right);
```

```
   }
}

main(){
   printf("Input Array: ");
   display();
   printline(50);
   quickSort(0,MAX-1);
   printf("Output Array: ");
   display();
   printline(50);
}
```

If we compile and run the above program then it would produce following result −

Output

```
Input Array: [4 6 3 2 1 9 7 ]
==================================================
 pivot swapped :9,7
Updated Array: [4 6 3 2 1 7 9 ]
 pivot swapped :4,1
Updated Array: [1 6 3 2 4 7 9 ]
 item swapped :6,2
 pivot swapped :6,4
Updated Array: [1 2 3 4 6 7 9 ]
 pivot swapped :3,3
Updated Array: [1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]
==================================================
```

Analysis of Quick sort

Best Case

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in $A[p \,..\, r]$ every time procedure 'Partition' is called. The procedure 'Partition' always split the array to be sorted into two equal sized arrays.

If the procedure 'Partition' produces two regions of size $n/2$. the recurrence relation is then

$T(n) = T(n/2) + T(n/2) + \Theta(n)$

$= 2T(n/2) + \Theta(n)$

And from case 2 of Master theorem

$T(n) = \Theta(n \lg n)$

Worst-case

Let $T(n)$ be the worst-case time for QUICK SORT on input size n. We have a recurrence

$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n)$ --------- 1

where q runs from 1 to n-1, since the partition produces two regions, each having size at least 1.

Now we guess that $T(n) \leq cn^2$ for some constant c.

Substituting our guess in equation 1.We get

$$T(n) = \max_{1 \leq q \leq n-1} (cq^2) + c(n - q^2)) + \Theta(n)$$

$$= c \max (q^2 + (n - q)^2) + \Theta(n)$$

Since the second derivative of expression $q^2 + (n-q)^2$ with respect to q is positive. Therefore, expression achieves a maximum over the range $1 \leq q \leq n-1$ at one of the endpoints. This gives the bound $\max (q^2 + (n - q)^2)) \; 1 + (n-1)^2 = n^2 + 2(n-1)$.

Continuing with our bounding of $T(n)$ we get

$$T(n) \leq c\,[n^2 - 2(n-1)] + \Theta(n)$$

$$= cn^2 - 2c(n-1) + \Theta(n)$$

Since we can pick the constant so that the $2c(n-1)$ term dominates the $\Theta(n)$ term we have

$$T(n) \leq cn^2$$

Thus the worst-case running time of quick sort is $\Theta(n^2)$.

STRASSEN'S MATRIX MULTIPLICATION

Given two square matrices A and B of size n x n each, find their multiplication matrix.

Naive Method

Following is a simple way to multiply two matrices.

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity of above method is $O(N^3)$.

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$T(N) = 8T(N/2) + O(N^2)$

From Master's Theorem, time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of Strassen's method is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divides matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

p1 = a(f - h) **p2 = (a + b)h**
p3 = (c + d)e **p4 = d(g - e)**
p5 = (a + d)(e + h) **p6 = (b - d)(g + h)**
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as $T(N) = 7T(N/2) + O(N^2)$.

From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Generally, Strassen's Method is not preferred for practical applications for following reasons.

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The sub matrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on non integer values, larger errors accumulate in Strassen's algorithm than in Naive Method

FINDING MAXIMUM AND MINIMUM

<u>Problem:</u> Finding the maximum and minimum elements in a set of (n) elements using the straightforward algorithm.

```
Algorithm straightforward (a, n, max, min)
    Input: array (a) with (n) elements
    Output: max: max value, min: min value
           max = min = a(1)
           for i = 2 to n do
           begin
                    if (a(i)>max) then max=a(i)
                    if (a(i)<min) then min=a(i)
           end
    end
```

best= average =worst= 2(n-1) comparisons

If we change the body of the loop as follows:

```
        max=min=a(1)
        for i=2 to n do
        begin
                  if (a(i)>max) then max=a(i)
                  else     if (a(i)<min) then min=a(i)
        end
```

Complexity

- best case: elements in increasing order No. of comparisons = (n-1)
- Worst case: elements in decreasing order No. of comparisons = 2(n-1)

- Average case: a(i) is greater than max half the time No. of comparisons= 3n/2 – 3/2

  ½ ((n-1)+(2n-2))

Divide and Conquer approach

Problem: is to find the maximum and minimum items in a set of (n) elements.

```
Algorithm MaxMin(i, j, max, min)
   input: array of N elements, i lower bound, j upper bound
   output: max: largest value
                min: smallest value.
if (i=j) then max=min=a(i)
else
          if (i=j-1) then
                  if (a(i)<a(j)) then
                          max= a(j)
                          min= a(i)
                    else
                           max= a(i)
                           min= a(j)
        else
                   mid=(i+j)/2
                   maxmin(i, mid, max, min)
                   maxmin(mid+1, j, max1, min1)
        if (max<max1) then max = max1
        if (min>min1) then min = min1
 end
```

Complexity Analysis

1.if size= 1 return current element as both max and min //base condition

2.else if size= 2 one comparison to determine max and min //base condition

3.else /* if size > 2 find mid and call recursive function */

recur for max and min of left half recur for max and min of right half.

one comparison determines to max of the two subarray, update max.

one comparison determines min of the two subarray, update min.

4. finally return or print the min/max in whole array.

Recurrence relation can be written as T(n)=2(T/2)+2 solving which give you T(n) =3/2n-2 which

is the exact no. of comparisons but still the worst time complexity will be T(n)=$O(n)$ and best case time complexity will be $O(1)$ when you have only one element in array, which will be candidate for both max and min.

$$C_n = \begin{cases} C\left(\lceil \frac{n}{2} \rceil\right) + C\left(\frac{n}{2}\rfloor\right) + 2 & for\ n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of 2, $n = 2^k$

$$\begin{aligned} C_n &= 2C_{\frac{n}{2}} + 2 \\ &= 2\left(2C_{\frac{n}{4}} + 2\right) + 2 \\ &= 4C_{\frac{n}{4}} + 4 + 2 \\ &\vdots \\ &= 2^{k-1}C_2 + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= \frac{3n}{2} - 2 \end{aligned}$$

Note:

No algorithm based on comparison can do less than this.

ALGORITHM FOR FINDING CLOSEST PAIR

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

The Brute force solution is O(n^2), compute the distance between each pair and return the smallest. We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy. In this post, a O(n x (Logn)^2) approach is discussed. We will be discussing a O(nLogn) approach in a separate post.

Algorithm

Following are the detailed steps of a O(n (Logn)^2) algortihm.

Input: An array of n points *P[]*

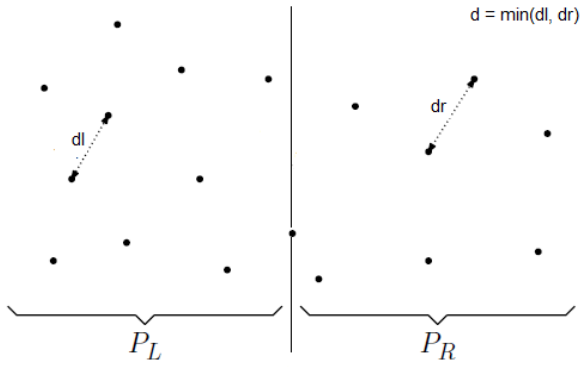
Output: The smallest distance between two points in the given array.

As a pre-processing step, input array is sorted according to x coordinates.

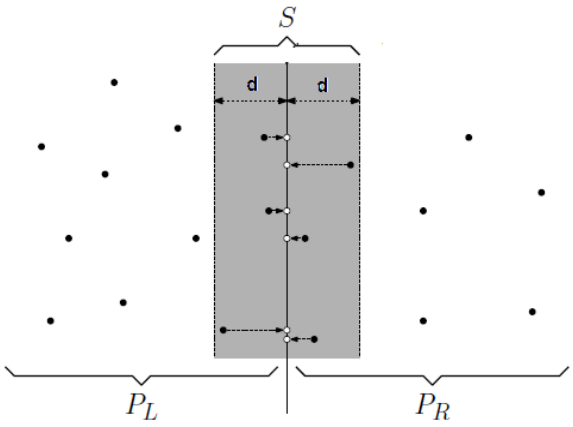
1) Find the middle point in the sorted array, we can take *P[n/2]* as middle point.

2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].

3) Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.



4) From above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. Consider the vertical line passing through passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.



5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.

6) Find the smallest distance in strip[]. This is tricky. From first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate). See this for more analysis.

7) Finally return the minimum of d and distance calculated in above step (step 6)

Time Complexity Let Time complexity of above algorithm be T(n). Let us assume that we use a O(nLogn) sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time, sorts the strip in O(nLogn) time and finally finds the closest points in strip in O(n) time. So T(n) can expressed as follows
T(n) = 2T(n/2) + O(n) + O(nLogn) + O(n)

T(n) = 2T(n/2) + O(nLogn)

T(n) = T(n x Logn x Logn)

Notes

1) Time complexity can be improved to O(nLogn) by optimizing step 5 of the above algorithm.
2) The code finds smallest distance. It can be easily modified to find the points with smallest distance.
3) The code uses quick sort which can be O(n^2) in worst case. To have the upper bound as O(n (Logn)^2), a O(nLogn) sorting algorithm like merge sort or heap sort can be used

CONVEX HULL PROBLEM

A polygon is **convex** if any line segment joining two points on the boundary stays within the polygon. Equivalently, if you walk around the boundary of the polygon in counterclockwise direction you always take left turns.

The **convex hull** of a set of points in the plane is the smallest convex polygon for which each point is either on the boundary or in the interior of the polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. A **vertex** is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are all vertices of the convex hull.

We discuss three algorithms:

- Graham Scan,
- Jarvis March and
- Divide & Conquer

We present the algorithms under the **assumption** that: no 3 points are collinear (on a straight line)

1. **Graham Scan**

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order.

Let $\mathbf{x}_0$ be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from $\mathbf{x}_0$ going counterclockwise: $\mathbf{x}_1; \mathbf{x}_2; : : : ; \mathbf{x}_{n-1}$. Let $\mathbf{x}_n = \mathbf{x}_0$, the chosen point. Assume that no two points have the same angle from $\mathbf{x}_0$.

The algorithm is simple to state with a single stack:

Graham Scan

1. Sort points by angle from $\mathbf{x}_0$
2. Push $\mathbf{x}_0$ and $\mathbf{x}_1$. Set **i**=2
3. While **i** <=**n** do:

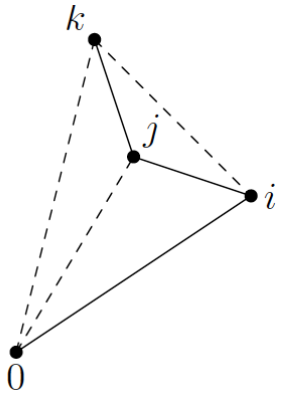
If $\mathbf{x}_i$ makes left turn w.r.t. top 2 items on stack, then

{ push **x**i; **i**++ }

else { pop and discard }

To prove that the algorithm works, it success to argue that:

- A discarded point is not in the convex hull. If $\mathbf{x}_j$ is discarded, then for some **i < j < k** the points **x**i --> **x**j --> **x**k form a right turn. So, $\mathbf{x}_j$ is inside the triangle $\mathbf{x_0}$, $\mathbf{x_i}$, $\mathbf{x_k}$ and hence is not on the convex hull.



- What remains is convex. This is immediate as every turn is a left turn.

The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most 2**n** times. There is a constant-time subroutine for checking, given three points in order, whether the angle is a left or a right turn. This gives an **O**(**n**) time algorithm, apart from the initial sort which takes time **O**(**n** log **n**). (Recall that the notation **O**(**f**(**n**)), pronounced "order **f**(**n**)", means "asymptotically at most a constant times **f**(**n**)".)

2. Jarvis March

This is also called the **wrapping algorithm**. This algorithm finds the points on the convex hull **in the order** in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many.

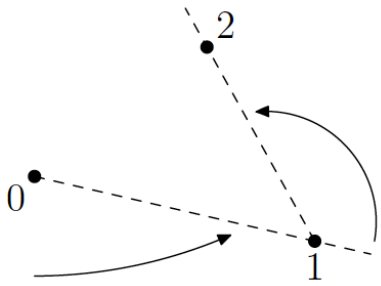
Let **x**0 be the leftmost point. Let **x**1 be the first point counterclockwise when viewed from**x**0. Then **x**2 is the first point counterclockwise when viewed from **x**1, and so on.

Jarvis March

i = 0

while not done do

$\mathbf{x}_{i+1}$ = first point counterclockwise from $\mathbf{x}_i$



Finding **xi**+1 takes linear time. The while loop is executed at most **n** times. More specifically, the while loop is executed **h** times where **h** is the number of vertices on the convex hull. So Jarvis March takes time **O**(**nh**).

The best case is **h** = 3. The worst case is **h** = **n**, when the points are, for example, arranged on the circumference of a circle.

3. Divide and Conquer

Divide and Conquer is a popular technique for algorithm design. We use it here to find the convex hull.

The first step is a **Divide step**, the second step is a **Conquer step**, and the third step is a **Combine step.**

The idea is to:

Divide and conquer

1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

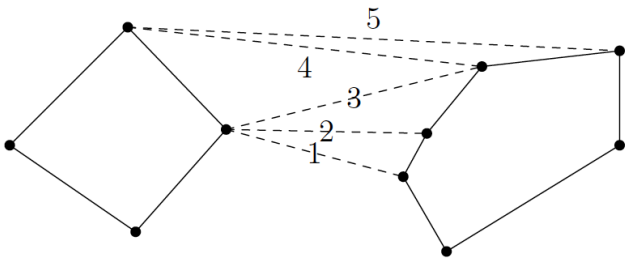
Combining two hulls

It helps to work with convex hulls that do not overlap. To ensure this, all the points are **presorted** from left to right. So we have a left and right half and hence a left and right convex hull.

Define a **bridge** as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the **upper** and **lower** bridges.

The following produces the upper bridge.

1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.
2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.
3. If made no progress in (2) (cannot raise either side), then stop else repeat (2).



We need to be sure that one will eventually stop.

Running time of the algorithm. The key is to perform step (2) in constant time. For this it is sufficient that each vertex has a pointer to the next vertex going clockwise and going counterclockwise. Hence the choice of data structure: we store each hull using a **doubly linked circular linked list**.

It follows that the total work done in a merge is proportional to the number of vertices. This means that the overall algorithm takes time **O**(**n** log **n**).

Introduction, Binary Search

A sorting technique is called stable if it

a) Takes O(nlogn) times

b) Maintains the relative order of occurrence of non-distinct elements

c) Uses divide-and-conquer paradigm

d) Takes O(n) space

Which one of the following option is true for binary search?

(A) It uses greedy approach

(B) It uses heuristic search

(C) It uses divide and conquer strategy

(D) It uses backtracking approach

We have a Max heap, which is represented by an array, and the process of inserting an element into it is going on. How many comparisons are done to find the position for the newly inserted element if we perform a binary search on the path from the new leaf to the root?

(A) θ (log2 n)

(B) θ (log2 log2 n)

(C) θ (log2 n/2)

(D) θ (n)

Binary search algorithm can not be applied to

a. **sorted linked list**

b. sorted binary trees

c. sorted linear array

d. pointer array

Running binary search on an array of size n which is already sorted is

a) O(nlogn)

b) O(logn)

c) $O(n^2)$

d) $O(n/2)$

Which of the following is not a limitation of binary search algorithm?

A. must use a sorted array

B. requirement of sorted array is expensive when a lot of insertion and deletions are needed

C. there must be a mechanism to access middle element directly

D. binary search algorithm is not efficient when the data elements more than 1500.

If the recurrence relation for a binary search is represented by T(n)=aT(n/b)+O(n^d) then the value of a+b+d is *

1. **3**
2. 4
3. 1
4. 0

Which of the following algorithm is of divide and conquer type?

A. Bubble sort

B. Insertion sort

C. **Binary search**

D. Closest pair

Which of the following is not the required condition for binary search algorithm?

A. The list must be sorted

B. There should be the direct access to the middle element in any sub list

C**. There must be mechanism to delete and/or insert elements in list.**

D. Number values should only be present

Finding the location of a given item in a collection of items is called .....

A. Discovering

B. Finding

C. Searching

D. Mining

Merge sort and its algorithm analysis

Which one of the below is not divide and conquer approach?

1. Insertion Sort
2. **Merge Sort**
3. Shell Sort
4. Heap Sort

Which of the following sort algorithms are guaranteed to be O(nlog n) even in the worst case?

1. Shell Sort
2. Quick Sort
3. **Merge Sort**
4. Insertion Sort

Which of the following is a stable sorting algorithm?

a) Merge sort

b) Typical in-place quick sort

c) Heap sort

d) Selection sort

Which of the following is not an in-place sorting algorithm?

a) Selection sort

b) Heap sort

c) Quick sort

d) Merge sort

Running merge sort on an array of size n which is already sorted is

a) O(n)

b) O(nlogn)

c) $O(n^2)$

d) None

Which of the following sorting algorithms has the lowest worst-case complexity?

1. **Merge sort**
2. Bubble sort
3. Insertion Sort
4. Quick Sort

When the merge sort is implemented iteratively then which of the following data structures is used *

1. Stacks
2. **Queues**
3. Graphs
4. Tree

Merge sort uses

a) Divide-and-conquer

b) Backtracking

c) Heuristic approach

d) Greedy approach

For merging two sorted lists of size m and n into sorted list of size m+n, we require comparisons of

a) O(m)

b) O(n)

c) O(m+n)

d) O(logm + logn)

The operation that combines the element is of A and B in a single sorted list C with n=r+s element is called ....

A. Inserting

B. Mixing

C. Merging

D. Sharing

A list of n string, each of length n, is sorted into lexicographic order using the merge-sort algorithm.

The worst case running time of this computation is

1. O(nlogn)
2. **$O(n^2 logn)$**
3. $O(n^2)$
4. $O(n^2 + logn)$

Consider the Quicksort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let T(n) be the number of comparisons required to sort n elements. Then

1. $T(n) <= 2T(n/5) + n$
2. **$T(n) <= T(n/5) + T(4n/5) + n$**
3. $T(n) <= 2T(4n/5) + n$

$T(n) <= 2T(n/2) + n$

Quick sort and its algorithm analysis

Which of the following is not a stable sorting algorithm in its typical implementation.

1. Insertion Sort
2. Merge Sort
3. **Quick Sort**
4. Bubble sort

You have an array of n elements. Suppose you implement quicksort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is

1. **O(n^2)**
2. O(nLogn)
3. Theta(nLogn)
4. O(n^3)

Which one of the following is the recurrence equation for the worst case time complexity of the Quicksort algorithm for sorting n(≥ 2) numbers? In the recurrence equations given in the options below, c is a constant.

A T(n) = 2T (n/2) + cn
B T(n) = T(n – 1) + T(0) + cn
C T(n) = 2T (n – 2) + cn
D T(n) = T(n/2) + cn

Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this: 2 5 1 7 9 12 11 10 Which statement is correct?

1. The pivot could be either the 7 or the 9.
2. The pivot could be the 7, but it is not the 9
3. The pivot is not the 7, but it could be the 9
4. Neither the 7 nor the 9 is the pivot.

The time complexity of quick sort is .....

A. O(n)

B. O(logn)

C. O(n2)

D. O(n logn)

Quick sort is also known as .....

A. merge sort

B. tree sort

C. shell sort

D. partition and exchange sort

Given the following list of numbers [14, 17, 13, 15, 19, 10, 3, 16, 9, 12] which answer shows the contents of the list after the second partitioning according to the quicksort algorithm?

1. [9, 3, 10, 13, 12]
2. [9, 3, 10, 13, 12, 14]
3. [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
4. **[9, 3, 10, 13, 12, 14, 19, 16, 15, 17]**

A pivot element to partition unsorted list is used in

1. Merge sort
2. **Quick Sort**
3. Selection sort
4. Insertion sort

Which of the following algorithm design technique is used in the quick sort algorithm?

a) Dynamic programming

b) Backtracking

c) Divide-and-conquer

d) Greedy method

Partition and exchange sort is .....

A. quick sort

B. tree sort

C. heap sort

D. bubble sort

Which of the following sorting algorithm is of divide and conquer type?

A. Bubble sort

B. Insertion sort

C. Quick sort

D. Merge sort

What is recurrence for worst case of QuickSort and what is the time complexity in Worst case?
Recurrence is T(n) = T(n-2) + O(n) and time complexity is O(n^2)
Recurrence is T(n) = T(n-1) + O(n) and time complexity is O(n^2)
Recurrence is T(n) = 2T(n/2) + O(n) and time complexity is O(nLogn)
Recurrence is T(n) = T(n/10) + T(9n/10) + O(n) and time complexity is O(nLogn)

Which is the correct order of the following algorithms with respect to their time Complexity in the best case ?

A Merge sort > Quick sort >Insertion sort > selection sort

B insertion sort < Quick sort < Merge sort < selection sort

C Merge sort > selection sort > quick sort > insertion sort

D Merge sort > Quick sort > selection sort > insertion sort

Explanation:

```
In best case,

Quick sort: O (nlogn)
Merge sort: O (nlogn)
Insertion sort: O (n)
Selection sort: O (n^2)
```

Strassen's Matrix multiplication

Finding Maximum and minimum

Algorithm for finding closest pair

Convex Hull Problem